ARMY RESEARCH LABORATORY

Comments on Computational Fluid Dynamics (CFD) Code Performance on Scalable Architectures

by Marek Behr, Daniel M. Pressel,
and Walter B. Sturek, Sr.

ARL-RP-63 December 2002

A reprint from *Computer Methods in Applied Mechanics and Engineering*, vol. 190, pp. 263–277, 2000.

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5067

ARL-RP-63 **December 2002**

Comments on CFD Code Performance on Scalable Architectures

Marek Behr
Rice University

Daniel M. Pressel and Walter B. Sturek, Sr.
Computational Information and Sciences Directorate, ARL

A reprint from *Computer Methods in Applied Mechanics and Engineering*, vol. 190, pp. 263–277, 2000.

Comments on CFD Code Performance on Scalable Architectures

Marek Behr
Mechanical Engineering and Materials Science
Rice University - MS 321, 6100 Main Street
Houston, TX 77005, USA

Daniel M. Pressel and Walter B. Sturek, Sr.
U.S. Army Research Laboratory
Aberdeen Proving Ground, MD 21005, USA

July 14, 1999
Revised September 22, 1999

Abstract

We comment on the current performance of computational fluid dynamics codes on a variety of scalable computer architectures. The performance figures are derived from both the finite volume and finite element methodologies, and encompass shared, virtual shared, and distributed memory architectures, as exemplified by the SGI Origin series, CM5, and the CRAY T3D/E family, respectively.

1. Introduction

The purpose of this paper is to provide an overview of recent efforts to obtain high levels of computer code performance using scalable computer architectures, while performing Computational Fluid Dynamics (CFD) modeling of challenging physical applications. The scope of this paper includes experiences using finite-difference/finite-volume and finite-element techniques. This paper considers only a limited number of computer techniques and computer architectures so there is no intent for this paper to be considered as a definitive evaluation. Rather, the intent is to gather the experience of a limited set of researchers and provide a summary of their experiences, in order to define the current state-of-the-art for scalable computing from these researchers perspective. The paper first outlines the approaches available in order to maximize the performance of a CFD code, discusses issues and experiences with finite-difference/finite-volume (FD/FV) codes in Section 3 and then addresses similar issues for finite-element (FE) codes in Section 4. Two implementations of computational fluid dynamics codes on distributed memory architectures are discussed and analyzed for scalability.

2. Optimization Issues

We start by discussing optimization issues that have the greatest impact on code performance, starting with loop-level parallelism and serial efficiency, and ending with the coarse-level parallelism, which is typically exploited on distributed-memory architectures.

2.1. Loop-Level Parallelism

Certain classes of numerical methods are known to be inherently difficult to parallelize using standard techniques on commonly available distributed-memory Multiple-Instruction, Multiple-Data (MIMD) computers. One such class of problems involve *implicit* CFD codes, represented in this paper by the work done by the authors and their colleagues on the ZNS-FLOW code. However, if a code is known to be vectorizable, then in theory it can be parallelized using some form of *loop-level parallelism.* This statement is based on the fact that vectorization is in fact a form of loop-level parallelism.

Having said this, one has to consider why this technique is not used more often. The answer to this question is complicated and is beyond the scope of this paper. However, the following is a summary of a few of the issues involved:

1. Most efforts involving parallelization assume High Performance Computing (HPC) is synonymous with an almost unlimited amount of parallelism. While once this was the case, this is frequently no longer the case. This is important, since many 3D problems will have loops that only support a limited level of loop-level parallelism (e.g., 100 iterations).

2. Loop-level parallelism is easiest to implement using compiler directives. Unfortunately, many systems (especially distributed memory systems) either lack these directives entirely, and/or have particularly inefficient implementations of these directives.

3. While many vector computers do support compiler directives for loop-level parallelism, there might not be enough available parallelism to show both efficient vectorization and parallelization. Even in cases where things worked well in theory, the combination of the limited number of processors on vector machines and the usage policies at most sites made it difficult to show even moderate levels of parallel speedup.

4. Most of the remaining machines that did support compiler directives for loop-level parallelism, are/were either too small, too weak, had too little memory, or in general were in some way too limited to be of much use for High Performance Computing.

But gradually, as the data presented in this paper here suggests, systems that will meet the needs of HPC when using loop-level parallelism are becoming a reality. The one remaining piece of the puzzle is serial efficiency, and it will be discussed in the next section.

2.2. Serial Efficiency

Those who are used to running highly vectorizable programs on high-end vector computers (e.g., the CRAY C90) are used to achieving relatively high levels of efficiency (e.g., 30 percent or better of peak). On the other hand, those who are used to running programs of systems equipped with Reduced Instruction Set Computing (RISC) processors and memory hierarchies involving cache, frequently report low levels of efficiency even for serial code (e.g., numbers in the range of 1–5 percent are frequently mentioned). Clearly, if one starts out with a low level of serial efficiency, a reasonable assumption is that when the code is parallelized,

the overall level of efficiency will drop even further. Therefore, if one is to use only moderate numbers of processors, it is important to use them as efficiently as possible.

As a result of these observations, a significant effort was made by the authors to tune the ZNSFLOW code for serial efficiency on systems using RISC processors. To a large extent, the success of this effort depended on the ability to take advantage of the cache. Since the amount of cache per processor ranged from 16 KB of on-chip data cache for the CRAY T3D to 1–4 MB of off-chip cache (either data or combined secondary cache, depending on the system) for the shared memory systems from SGI and Convex, it is probably not surprising to find out that these efforts were not equally successful on all machines. Even so, these efforts made a significant contribution to the success of this project.

2.3. Coarse-Grain Parallelism and Portability

To date, most methods for enabling loop-level parallelism have been tied to a specific architecture (CRAFT on CRAY Massively Parallel Processors or MPPs, C$DOACROSS directives on SGI Symmetric Multi-Processors or SMPs). There is a hope that the emerging OpenMP standard will make codes that exploit loop-level parallelism widely portable. Another approach which may be considered as a way of enhancing code performance and portability, is to rewrite the legacy code to exploit parallelism at a coarser level than the level of individual loops. This approach carries with it a large investment, but may be the only way of achieving parallel performance on multiple architectures with a single code base, by taking advantage of nearly universally available message-passing libraries, such as the Message Passing Interface (MPI) [1] or the Parallel Virtual Machine (PVM) [2]. This issue will be discussed in-depth in the context of two application examples in Section 5.

3. Finite Difference and Finite Volume Computational Techniques

Efficient parallel performance can be achieved with relative ease for explicit CFD solvers using domain decomposition. However, the real workhorse for Army Research Laboratory (ARL) application specialists in aerodynamics has been codes which have evolved from the Beam-Warming [3] linearized block implicit technique. In order to maintain familiar operational aspects of the codes in terms of implicit performance and basic code structure, it was desired to explore the parallelization of the code keeping the solver algorithm essentially intact. Implicit flow solvers are generally regarded as being computationally efficient. These techniques have, in the past, been considered difficult to convert and have perform efficiently on scalable architectures. Recent experience on RISC-Based Shared Memory SMPs has changed this perception. Sahu et al. [4] have shown that highly efficient performance can be achieved if one is willing to expend the effort to restructure the code (starting with a code structured for a vector computer architecture). Behr and Sahu [5] have obtained experience in porting the same code to a distributed memory architecture. This experience provides an interesting comparison to the SMP experience.

3.1. Shared Memory Architectures

Machines utilizing these architectures include SGI Power Challenge Array (PCA), SGI R10K PCA, SGI Origin 2000, Convex Exemplar, SGI Challenge, and SGI Origin 200. These multiprocessors feature powerful RISC processors with various levels and characteristics of cache memory in addition to the main memory. Loop-level parallelism has been implemented using compiler directives. Although tools exist to assist the programmer to identify areas of code that can benefit from modification and performance analysis, the technique used by Pressel [4] was to manually modify code and invoke timing queries to identify and evaluate modifications to the code. This process was very time consuming and required a dedicated computer scientist; however, the payoff was a high degree of performance and greatly increased knowledge of how the computer architecture features affect code performance. The effort required to achieve the performance level reported here was about 9 man-months. An example of the performance achieved for on Origin 2000 for a set of three problem sizes and different partitions of processing elements (PEs) is given in Table 1 and illustrated in Figure 1. The 1 million and 12 million cell results can be directly compared to the message passing results presented in Section 5. The results for the Origin 2000 show excellent scala-

PEs	1 million cells	12 million cells	59 million cells
8	793[1]	66[1]	12[2]
16	1407[2]	122[1]	22[2]
24	1895[2]	174[2]	28[2]
32	2138[2]	215[2]	34[2]
40	2736[2]	240[2]	51[2]
48	2725[2]	295[2]	58[2]
56	2862[2]	318[2]	65[2]
64	2601[3]	309[2]	73[3]
72	2911[3]	292[3]	72[3]
80	3420[3]	332[3]	79[3]
88	3619[3]	400[3]	86[3]
96	–	400[4]	92[4]
104	–	400[4]	92[4]
112	–	413[3]	102[3]
120	–	–	103[3]
CRAY C90	227	20[5]	–[6]

[1] Machine with 32 195 MHz R10K nodes.
[2] Machine with 64 195 MHz R10K nodes.
[3] Machine with 128 195 MHz R10K nodes (pre-production).
[4] Estimated from flat ≤ 92 and ≥ 109 measurements.
[5] Extrapolated.
[6] Problem too large to fit in available memory.

Table 1. Scalable performance of ZNSFLOW on SGI Origins in time steps per hour; CRAY C90 performance given for comparison.

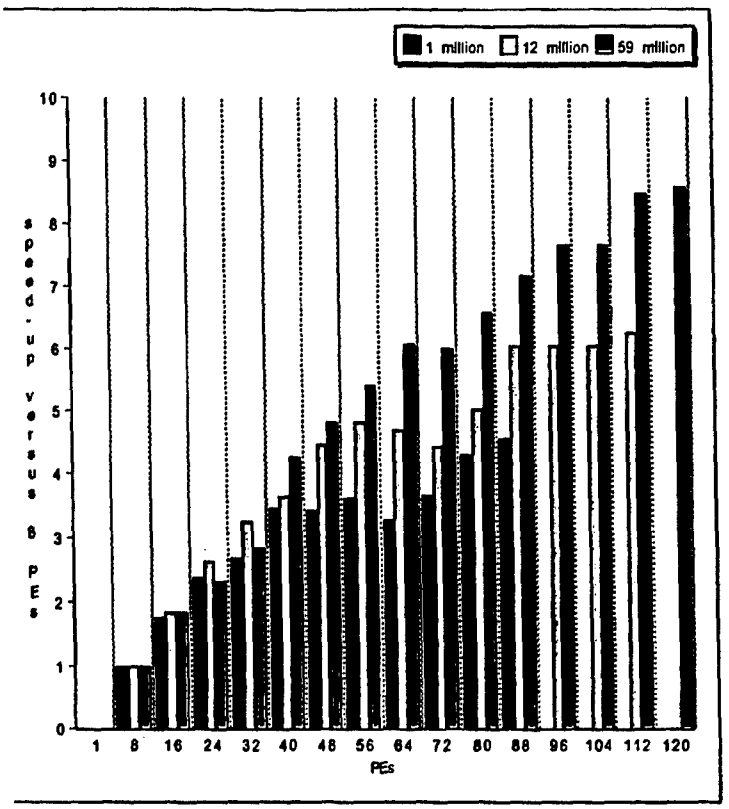


Figure 1. Graph of scalable performance of ZNSFLOW on SGI Origins relative to 8 PEs.

bility and performance for up to 30 processors. While results to use more than 30 processors on larger Origins have been largely successful, the exact results are highly dependent on the problem size. This is a direct result of using loop-level parallelism, and that for the smaller problems one can expect to run out of parallelism before one runs out of processors. For larger problems, this is not the case. However, even for very large problems, the performance can be limited when using more than 50 processors. The source of this limitation is our decision not to parallelize many of the boundary condition routines. As a result, the job spends about 0.4-0.8 percent of it's CPU cycles executing serial code. When using 100 processors, a code with 1 percent serial code will show at best a factor of 50 speedup. These results clearly show that the techniques employed enable highly efficient performance to be achieved.

The performance achieved in terms of Millions FLating-point OPerations per Second (MFLOPS) is summarized in Table 2. The ratio of performance achieved to peak performance is, in general, about 10 – 30% for the SMPs compared to the 30% value for the vector code on the Cray C90.

3.2. Distributed Memory Architectures

Coarse-grain parallelism was implemented on CRAY T3D and T3E architectures using the native SHMEM message-passing libraries on these computers. This strategy followed largely unsuccessful attempts at implementing loop-level parallelism using the virtual shared mem-

Machine	Type	PEs	Peak[1] MF	Actual[2] MF	% Peak[3]
CRAY C90	Vector	1	1,000	311	31.1
SGI Challenge	SMP	35	3,500	770	22.0
Origin 2000	SMP	30	11,700	2,958	25.3
Convex SPP1600	SMP	30	7,200	750	10.4
SGI PCA	SMP	18	5,400	1,033	19.1
SGI R10K PCA	SMP	15	5,850	1,369	23.4
CRAY T3D[4]	MIMD	128	19,200	1,080	5.6
CRAY T3E-600[4]	MIMD	49	29,400	1,400	4.8
CRAY T3E-1200[4]	MIMD	128	153,600	3,854	2.5

[1] Peak performance determined using FLOPS per clock cycle per processor.
[2] Actual performance achieved in MFLOPS.
[3] Percentage of peak performance actually achieved.
[4] Message-passing version.

Table 2. FV code performance results for selected architectures.

ory programming model (CRAFT). The compiler technology was found to be lacking in this task, leading to significantly lower-than-expected execution speeds, and high overheads for the automatically parallelized loops. Since improvements in this compiler technology were not immediately forthcoming, the decision was made to manually implement parallelism using message-passing code to control data motion. Significant changes from the serial code were required, with the programmer being responsible for all aspects of data distribution, inter-processor communication, and synchronization. This effort took approximately 6 man-months, and concentrated on producing scalable and portable code, with little scalar optimization work compared to the SMP version. The effectiveness of the compiler optimization of the code, at the level individual processors, remains lower than desired.

The performance achieved is indicated in Table 2 in terms of MFLOPS. Note that the scalability issues are addressed in Section 5. At the maximum partition sizes, after the code has been constrained by lack of available parallelism, the percentage of the peak performance is about 2–6%, well below that of the SMPs. Although the application was not optimally sized for the number of processors used on the CRAY MPP computers, the results achieved are believed to be in line with expectations.

4. Finite Element Techniques

Extensive experience with finite element CFD computations on parallel architectures has been gained over the past eight years by the Team for Advanced Flow Simulation and Modeling (TAFSM) at the Army HPC Research Center (AHPCRC) (see e.g. [6] and references therein). These computations are based on primitive and conservation variable formulations of Navier-Stokes equations of incompressible and compressible flows. They take advantage of implicit solvers based on iterative solution update techniques, such as Generalized Minimum RESidual (GMRES) [7]. These codes were implemented in the virtual shared memory model on the Connection Machine CM5 using the CM FORTRAN compiler (CMF), in the

shared memory model on the SGI Power Challenge (SMP), and in the distributed memory message-passing model (MP) on a range of architectures, including Cray T3D, Cray T3E, and IBM SP.

4.1. Virtual Shared Memory Architectures

In contrast with the experience discussed earlier in attempting to port the ZNSFLOW code to the virtual shared memory environment on the T3D, the experience with the AHPCRC FE codes using CMF on the Connection Machine has been largely positive, with unparalleled ease of use, and satisfactory performance. The finite element computations typically achieved 20 MFLOPS/node (of 120 peak MFLOPS/node) and demonstrated scalability up to 1024 nodes. However, further development of this particular architecture has since been abandoned.

4.2. Shared Memory Architectures

Finite element computations on the Power Challenge constitute a small portion of the overall research described in [6]. They employ loop-level parallelism via compiler directives. Typical computations exhibit speeds of 45 MFLOPS/node and are found to scale up to 12 processors (the maximum number available for these particular computations).

4.3. Distributed Memory Architectures

The dominant platform for the finite element computations described in [6] is a distributed memory architecture, with explicit message-passing calls facilitating data exchange between processors. Individual processing nodes are programmed using standard FORTRAN 77 and C languages; and data is exchanged using the MPI library. The cost of the initial porting effort (from serial or shared memory code) is later offset by excellent portability. These AHPCRC FE MPI-based codes have been moved with minimum of modifications between multiple architectures, such as Cray MPPs, SGI Origin, IBM SP or Sun HPC 10000. Typical computational speeds are 20 MFLOPS/node (of peak 150 MFLOPS/node) on CRAY T3D and 60-120 MFLOPS/node (of peak 1200 MFLOPS/node) on CRAY T3E-1200. Variations in computational speed can be observed depending on the variant of the GMRES algorithm being used. Matrix-based variant, which minimizes number of operations at the expense of memory, involves large data structures and large numbers of loads, and consequently invokes higher memory access costs. Matrix-free variant, used for large problems in order to minimize memory use, increases the total number of operations required, but results in smaller data structures and higher percentages of peak computational speed. Good scalability is observed on these platforms, with the possible exception of the SGI SMPs (see Section 5). These performance results are summarized in Table 3; they can be compared with the similar results for the FD/FV codes in Table 2.

5. Implementation Examples

Following the general discussion and results from loop-level parallelization in the previous sections, we now discuss in more detail implementation issues inherent in developing implicit

Machine	Type	PEs	Peak[1] MF	Actual[2] MF	% Peak[3]
CRAY C90	Vector	1	1,000	150	15.0
CM5	SIMD/MIMD	228	30,720	5,120	16.7
SGI PCA	SMP	12	3,600	540	15.0
CRAY T3D	MIMD	128	19,200	2,560	13.3
CRAY T3E-900	MIMD	49	44,100	6,370	14.4
CRAY T3E-1200[4]	MIMD	128	153,600	11,451	7.5

[1] Peak performance determined using FLOPS per clock cycle per processor.
[2] Actual performance achieved in MFLOPS.
[3] Percentage of peak performance actually achieved.
[4] Matrix-based version.

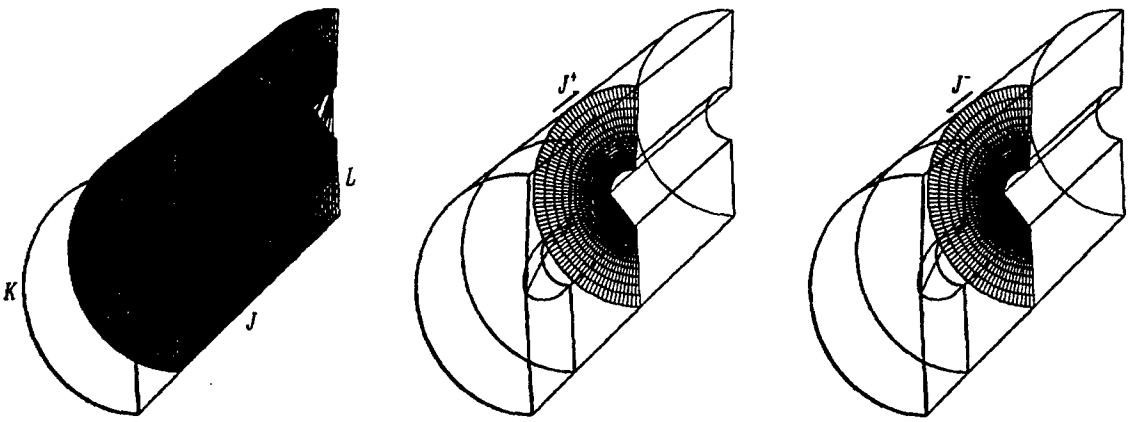
Table 3. FE code performance results for selected architectures.

codes that exploit coarse-grain parallelism. The two example implementations, one each of a finite volume code and a finite element code, take advantage of standard message-passing libraries, enhancing their portability.

5.1. Finite Volume Implementation on Distributed-Memory Machines

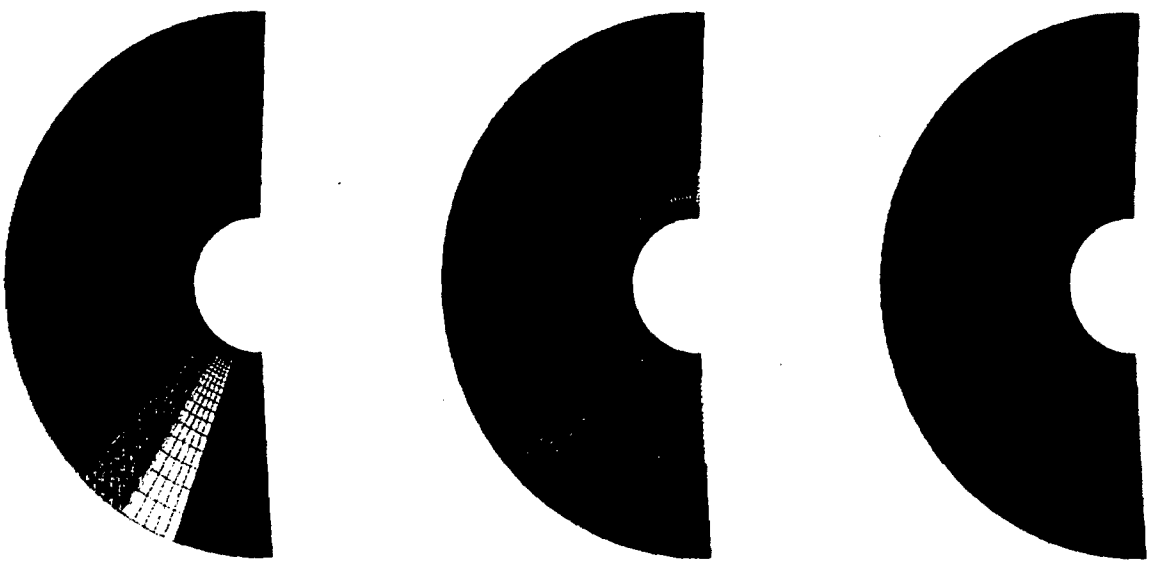
We start with the ZNSFLOW-D, the message-passing implementation of the ZNSFLOW finite volume code referred to in Section 3. In order to better explain the parallelization issues, an overview of the typical ZNSFLOW computation steps is given below. The computational grid is composed of multiple structured grid zones. All operations proceed on a zone-by-zone basis, with inactive zone data stored either in memory, or on a fast mass storage device. A single zone is constructed of a regular $NJ \times NK \times NL$ block of cells aligned with J, K, and L directions. The J direction is assumed to be streamwise, and is treated semi-implicitly with two solver sweeps in the J^+ and J^- directions. During the J^+ sweep, for each consecutive streamwise plane, the grid points are coupled in the L direction, while they are treated independently in the K direction. This requires a solution of K tridiagonal systems of size L with 5×5 blocks. In the J^- sweep the roles are reversed, with the coupling present in K direction only, and L block-tridiagonal systems of size K. Before the sweeping can commence, a volume calculation of the right hand side (RHS) must take place (Figure 2). An efficient parallel implementation of these two distinct computation phases, RHS formation, and solver sweeps, is crucial to the overall effectiveness and scalability of the code.

Out of the two computation-intensive stages of the code, the RHS formation yields itself to parallelization most easily. This is a volume computation, where each grid point is operated on independently, with only older values at neighboring points being required to complete the computation. The entire set of zone cells can be distributed over the available PEs in an arbitrary manner. However, for the sake of subsequent solver computations, it makes sense to decompose only K and L grid dimensions, leaving an entire J dimension associated with a single PE. The $K-L$ plane is mapped onto a rectangular grid of all PEs. To avoid repetition of inter-processor transfers, each rectangular portion of the $K-L$ plane also contains 2



a) RHS volume computation b) J^+ sweep c) J^- sweep

Figure 2. Data orientation and activity during ZNSFLOW-D phases for the last zone of the benchmark problem.



a) J^+ sweep b) RHS volume computation c) J^- sweep

Figure 3. Data distribution during ZNSFLOW-D phases.

layers of "ghost" points which track the two closest sets of values in the subgrids belonging to neighboring PEs. The parallelization of the solver sweeps is not as straightforward. The algorithm requires sequential processing in the J direction, and can also be simultaneously parallelized in both $K - L$ directions only at a great added computation cost, e.g., via a cyclic reduction algorithm. An alternative method is to accept serial treatment of the J and L directions (J and K for J^- sweep), and devote all PEs to parallelizing the K dimension (L for J^- sweep). This approach has the obvious disadvantage, since the scalability is not maintained as the number of PEs exceeds either the NK or NL zone dimensions. In typical computations however, the number of PEs and the zone dimensions are matched so that the problem does not arise. Therefore, for the solver sweeps the desired data distribution has the entire J and L dimensions associated with a single PE, and the K dimension decomposed

among all available PEs for the J^+ sweep; and J and K dimensions associated with a single PE and the L dimension distributed for the J^- sweep. This requires repeated reshaping of a small number of arrays between the original and two solver-specific layouts (Figure 3). A number of smaller parallelization issues had to be resolved as well, including parameter reading and broadcasting them among PEs, efficient disk I/O, and exchange of boundary data between zones.

As we mentioned in Section 3, the initial attempt to port the ZNSFLOW code to a scalable architecture involved the CRAY T3D and CRAFT shared memory programming model. The advantages of code maintainability and ease of transition were offset by the poor performance, and alternative approaches were explored. The more difficult task of rewriting the code in a message-passing framework was undertaken, and the PVM-based code provided initial speed-ups. The reshaping of the arrays during solver sweeps however was a difficult target for efficient implementation when using two-sided PVM communication. A much better solution was found in the form of the one-sided SHMEM CRAY communication libraries. In addition to eliminating concerns about deadlocking, the use of SHMEM reduces message latency and increases bandwidth. Apart from the communication issues, some scalar optimization of the code was attempted in order to extract a reasonable fraction of peak speed on cache-constrained architectures, but that aspect still leaves something to be desired. A variant based on the MPI library has been since added to the code base in order to ensure portability to platforms that do not support SHMEM, such as IBM SP and Sun HPC. It is anticipated that both the SHMEM and MPI portions will be replaced with a single one-sided MPI-2 version as this standard becomes widely accepted.

Speed and scalability of the message-passing code is tested on three architectures, using Mach 1.8 flow past an ogive cylinder at a 14° angle of attack on a 3-zone 1 million cell coarse grid, and the same geometry at Mach 2.5 on a 10 million cell fine grid. The results are listed in terms of time steps per hour in Tables 4 and 5, and also shown in graphical form in Figures 4 and 5. The CRAY T3E and SGI Origin platforms use the SHMEM-based version of ZNSFLOW-D, while the IBM SP employs the less efficient MPI-based version. For comparison, the CRAY C90 version of the code achieved 227 time steps per hour for the 1 million cell case. As expected, the plots show better scalability for the refined grid than for the coarse one, as parts of the current implicit solver contain parallelism only of the order of K or L dimensions. These dimensions are 75 and 70, respectively, for the coarse grid, and 180 and 140 for the refined one. The graphs exhibit visible notches around 70 and 75 PEs for the coarse grid, and around 70 PEs for the fine grid; these are thresholds at which the integer number of cells per PE (for the loops with K or L parallelism) decreases by one. A number of predictable secondary gradients in performance occurs as the integer number of cells per PE changes for the $K - L$ layouts. An example Mach number field at the conclusion of the 1 million cell simulation is shown in Figure 6.

PEs	T3E-1200	O2K (300 MHz)	SP (160 MHz)
8	349	382	199
16	616	618	288
24	888	838	335
32	1062	882	342
40	1324	989	374
48	1431	1083	420
56	1642	1161	428
64	1705	1050	423
72	2141	1326	405
80	2280	1382	420
88	2443	1320	396
96	2478		
104	2673		
112	2711		
120	2914		
128	2948		

Table 4. Scalable performance of ZNSFLOW-D on several platforms in time steps per hour for 1 million cell case.

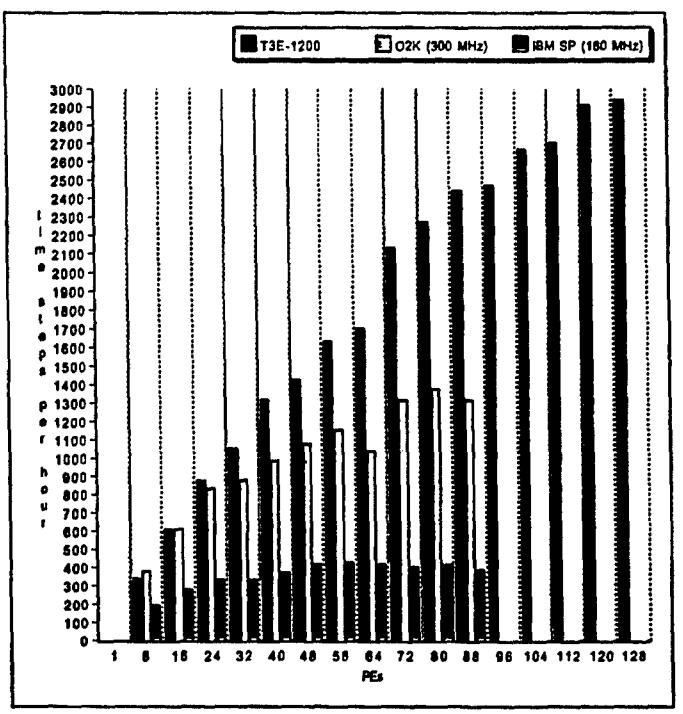


Figure 4. Graph of scalable performance of ZNSFLOW-D on several platforms in time steps per hour for 1 million cell benchmark case.

PEs	T3E-1200	O2K (300 MHz)	SP (160 MHz)
16	70		41
24	99	84	54
32	127	97	62
40	152	113	72
48	179	142	81
56	190	134	84
64	203	133	89
72	248	158	93
80	247	157	94
88	276	153	95
96	298		
104	317		
112	337		
120	355		
128	327		

Table 5. Scalable performance of ZNSFLOW-D on several platforms in time steps per hour for 10 million cell case.

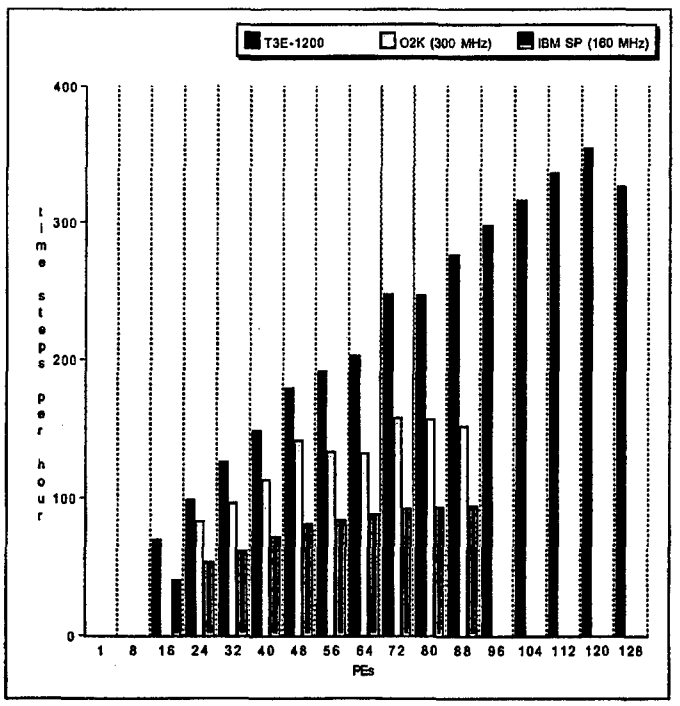


Figure 5. Graph of scalable performance of ZNSFLOW-D on several platforms in time steps per hour for 10 million cell case.

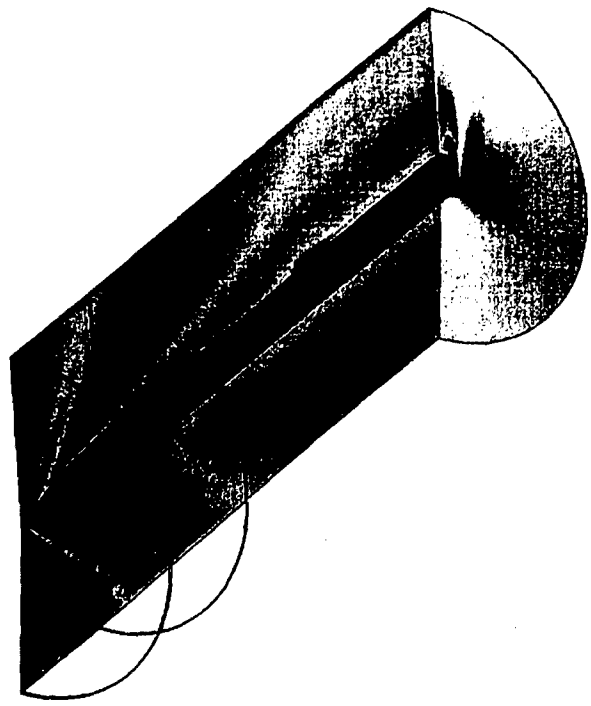


Figure 6. Ogive cylinder: Mach number distribution for the 1 million cell case.

5.2. Finite Element Implementation on Distributed-Memory Machines

The XNS code is based on a space-time variational formulation of the incompressible Navier-Stokes equations and its finite element discretization. It has evolved from a family of codes used in the late 80's on the CRAY C90, then ported to Connection Machine Fortran in early 90's, and finally rewritten for a message passing environment. This family of codes has been used to simulate a variety of fluid flows of engineering interest [6], including free-surface flows around hydraulic structures, various stages of parafoil descent, and flows around paratroopers egressing a cargo aircraft.

In contrast to ZNSFLOW-D, the XNS is designed to take advantage of unstructured finite element grids, and results in more complex data distribution. The structure of the code is essentially similar to the one previously described in the context of the Connection Machine [8, 9]. The finite element mesh is explicitly partitioned into a number of element sets, which form contiguous subdomains, with the aid of the METIS graph partitioning package [10]. With each set of elements assigned to a single processor, the nodes are then distributed in such a way that most nodes which are interior to a subdomain, are assigned to the processor which holds elements of the same subdomain. Nodes at a subdomain boundary are randomly assigned to processors sharing that boundary. The formation of the element-level components of the system of equations proceeds in the embarrassingly parallel fashion, with all data related to a given element residing on the same processor. The solution of that equation system takes place within a GMRES iterative solver. As described in [9], it is here that the bulk of inter-processor communication takes place, with the element-based structures (stiffness matrices, local residuals) interacting with node-based structures

(global residuals and increments). Movement of data from element-level to node-level takes the form of a *scatter*, and the reverse movement from element-level to node-level takes the form of a *gather*. Similarly to the Connection Machine Scientific Software Library (CMSSL) implementation, these operations take part in two stages: one local to the subdomain and free of communication, and another at the surface of the subdomains, involving moderate amount of communication. Similarly to ZNSFLOW-D, the XNS takes advantage of SHMEM libraries on the architectures that support them, and falls back to standard MPI on the ones that do not.

A typical application of XNS is a free-surface flow past a spillway of a dam, with the computational domain defined in Figure 7. The flow enters the domain through the inflow boundary (upper right), and proceeds past the spillway and the stilling basing with underwater energy dissipators, on to the outflow boundary (lower left). The upper surface is free to move according to the kinematic conditions at the fluid surface, and the lateral boundaries impose a periodicity condition, so that wider section of the dam may be represented with a narrow computational domain. The finite element mesh with 418,249 tetrahedral elements

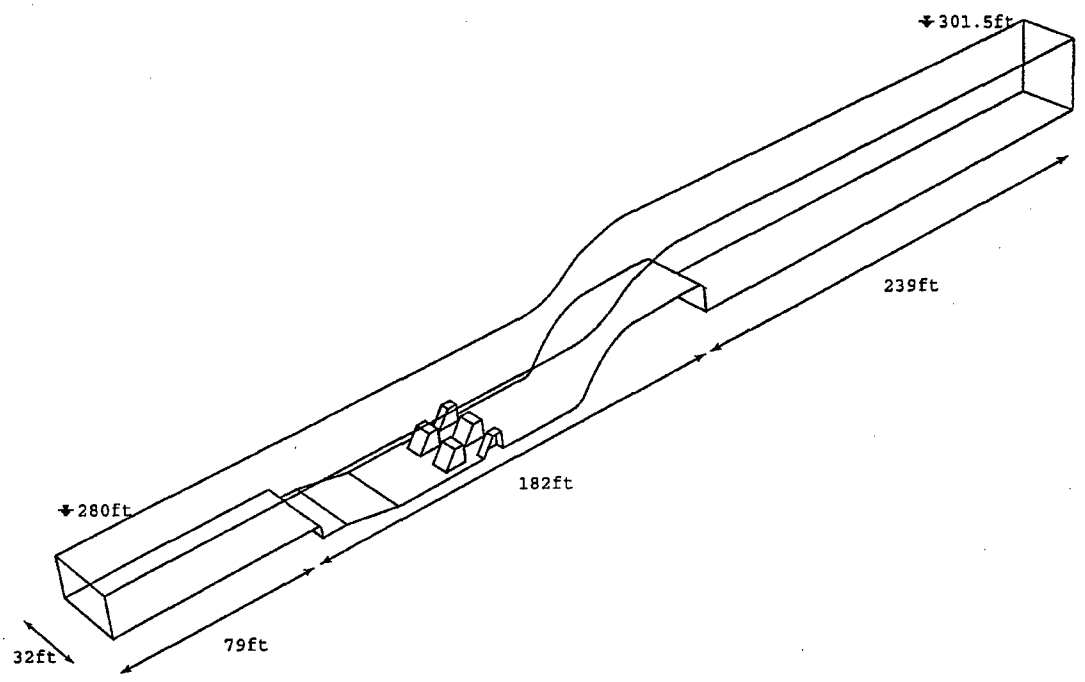


Figure 7. Flow in a spillway of a dam: computational domain.

is shown in Figure 8.

This simulation has been performed on several architectures and many partition sizes, in order to assess the portability and scalability of the XNS code. The results in terms of time steps per hour for CRAY T3E, SGI Origin and IBM SP platforms are listed in Table 6, and graphically presented in Figure 9. Owing to the large amount parallelism inherent in the finite element implementation, the code achieves almost perfect scalability on the CRAY T3E platform. The IBM SP takes a performance lead on small partitions, due to efficient library implementations of matrix-vector products and other primitives used in the finite element

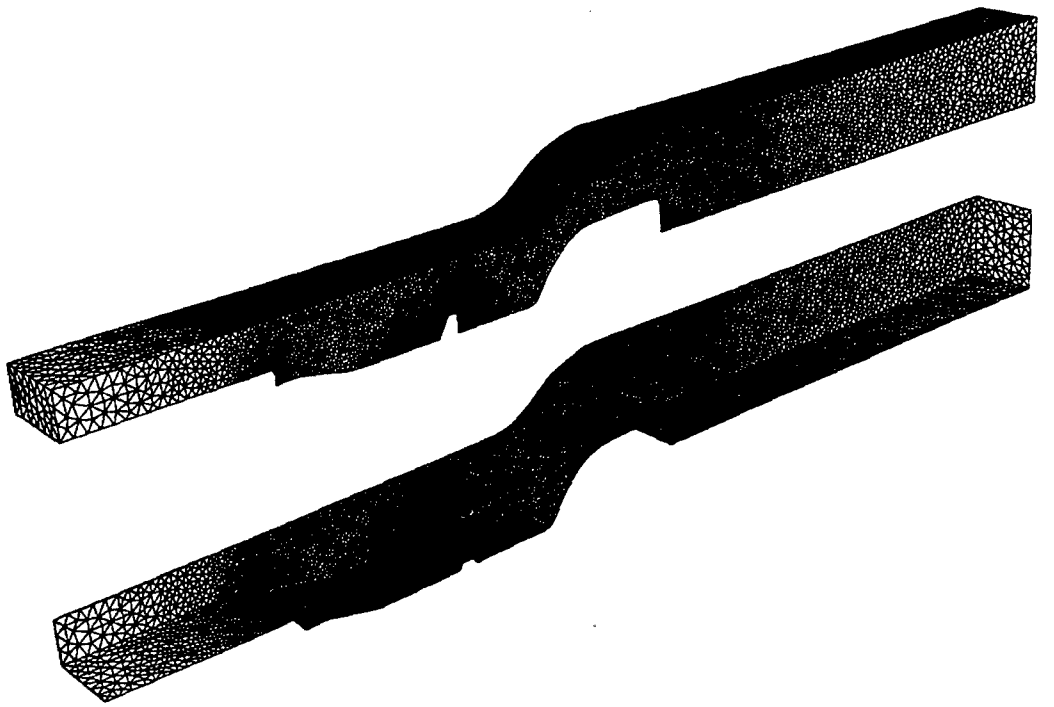


Figure 8. Flow in a spillway of a dam: finite element mesh.

code. However, the scalability on largest partitions of the SP starts to degrade, possibly due to higher latencies involved in the MPI communication, as compared to SHMEM. The Origin 2000 exhibits poor scalability in comparison; the cause has yet to be investigated.

The example flow field and the quasi-steady position of the free surface is shown in Figure 10, with the colors representing the streamwise component of the velocity field. For more details about free-surface simulations of this kind, interested reader is referred to [11].

PEs	T3E-1200	O2K (300 MHz)	SP (160 MHz)
16	6.3		10.1
24	9.3		14.4
32	12.7	8.4	18.2
40	15.4	9.8	22.0
48	18.5	11.3	24.4
56	21.6	13.2	26.0
64	25.1	12.4	29.9
72	27.5	11.8	30.5
80	30.5	14.0	31.4
88	34.1	14.2	31.8
96	36.5		
104	39.1		
112	42.1		
120	44.9		
128	48.9		

Table 6. Scalable performance of XNS on several platforms in time steps per hour.

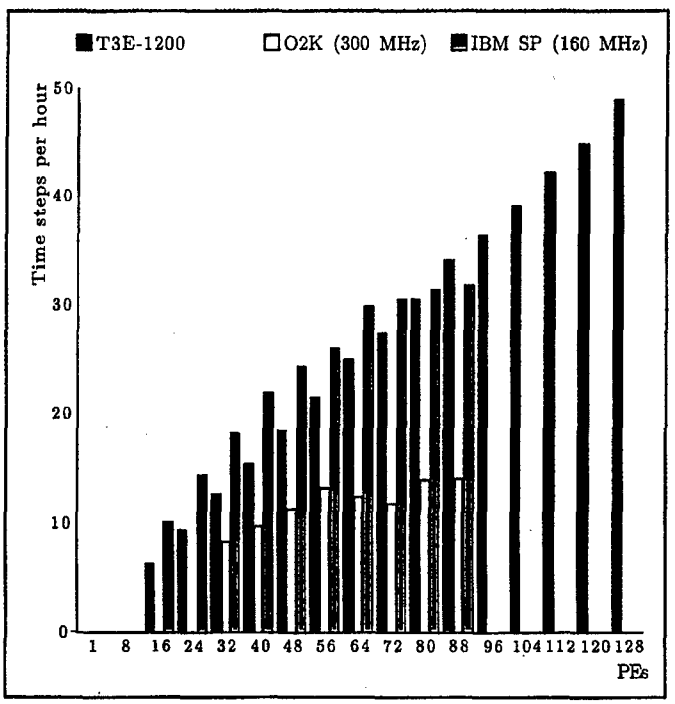


Figure 9. Graph of scalable performance of XNS on several platforms in time steps per hour.

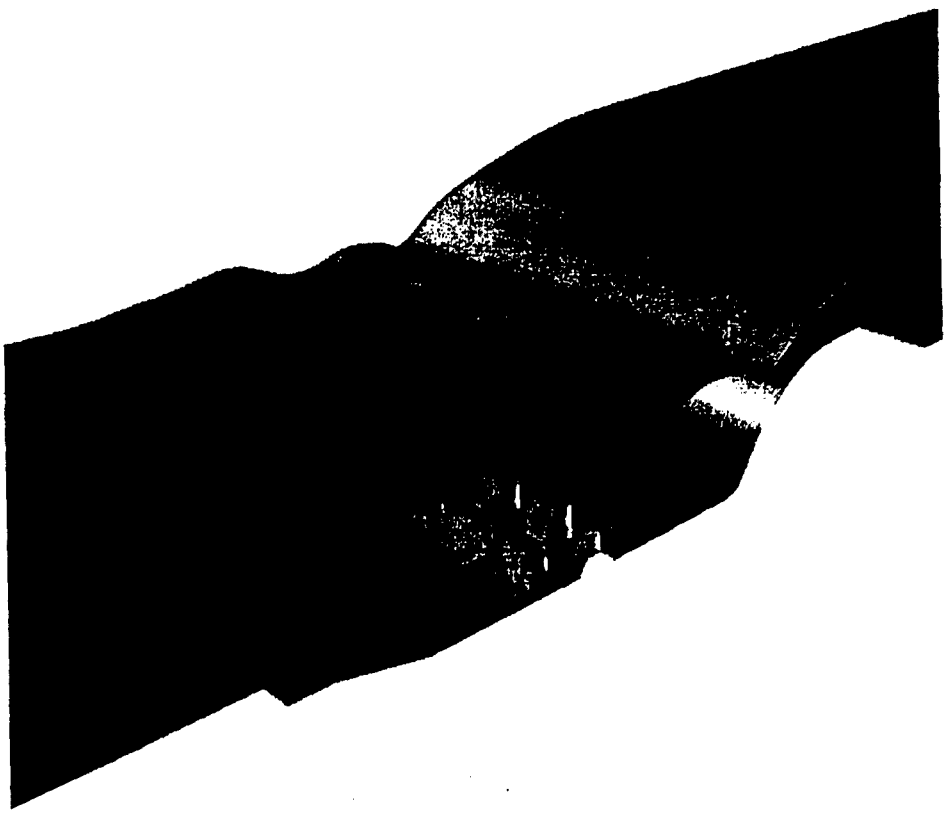


Figure 10. Flow in a spillway of a dam: free surface position and streamwise component of velocity at the spillway surface (slip boundary condition is employed).

6. Closing Remarks

The scalable performance achieved using a variety of parallel architectures, and two families of CFD codes, has been discussed. The results indicate that the percentage of peak performance achieved for our FD/FV computational codes is around 20–25% on SMP computers and about 2–6% for MIMD computers. The percentage of peak performance achieved for FE computational codes is 7–17% for SMP and MIMD computers. The level of programming effort required to achieve this performance is 6–9 man months when starting from a code that is written for a vector (CRAY C90) computer. There are many features of the computer architectures that affect the performance to be obtained, and also numerous programming techniques to be utilized. Table 7 presents some general features of the computer chip architectures for the results reported in this paper. In general, the SMP computer chips with the larger off-chip cache achieve the best performance. The FD/FV code performance on the T3D/T3E computers is penalized by the lack of a large off-chip-cache.

As computer chip technology continues to evolve, and compilers become available with more automatic features, it may be possible to achieve high levels of performance on scalable machines with less programmer effort. However, it appears that compiler technology

Machine	Peak MFLOPS per PE	Processor type	Data on-chip cache	off-chip cache	Comments
C90	1,000	Vector	–	–	[1]
CM5	120	Vector	–	–	[1]
T3D	150	RISC	16 KB	–	
T3E-900	900	RISC	8+96 KB	–	[2]
T3E-1200	1,200	RISC	8+96 KB	–	[2]
SGI	100	RISC-4400	16 KB	1-4 MB	[3]
SGI	300	RISC-8000	–	4 MB	[3]
SGI	390	RISC-R10K	32 KB	1-4 MB	[3]
SGI	600	RISC-R12K	32 KB	8 MB	[3]
IBM SP	480	RISC-P2SC	128 KB	–	[4]
Convex	240	RISC	–	1 MB	

[1] Vector registers, SRAM (very expensive and fast) memory.
[2] Streams.
[3] Moderate cache line size (128 bytes).
[4] Long cache line size (256 bytes).

Table 7. Computer Architecture Characteristics.

will continue to lag behind and significant hand-tuning of code will be required to achieve acceptable performance for the next several years.

7. Acknowledgement

This work was sponsored by the Army High Performance Computing Research Center under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008. The content does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred.

References

[1] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI.* MIT Press, Cambridge, Massachussetts, 1995.

[2] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manachek, and V. Sunderam, *PVM: Parallel Virtual Machine.* MIT Press, Cambridge, Massachussetts, 1994.

[3] T. Pulliam and J. Steger, "On implicit finite-difference simulations of three-dimensional flow", *AIAA Journal,* **18** (1992) 159–167.

[4] J. Sahu, D. Pressel, K. Heavey, and C. Nietubicz, "Parallel application of a Navier-Stokes solver for projectile aerodynamics", in *Proceedings of Parallel CFD'97,* Manchester, UK, (1997).

[5] M. Behr and J. Sahu, "Parallelization of a zonal missile aerodynamics code", *AHPCRC Bulletin*, **7** (1997) 9–12.

[6] T.E. Tezduyar, S. Aliabadi, M. Behr, A. Johnson, V. Kalro, and M. Litke, "Flow simulation and high performance computing", *Computational Mechanics*, **18** (1996) 397–412.

[7] Y. Saad and M. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems", *SIAM Journal of Scientific and Statistical Computing*, **7** (1986) 856–869.

[8] M. Behr, A. Johnson, J. Kennedy, S. Mittal, and T.E. Tezduyar, "Computation of incompressible flows with implicit finite element implementations on the Connection Machine", *Computer Methods in Applied Mechanics and Engineering*, **108** (1993) 99–118.

[9] J.G. Kennedy, M. Behr, V. Kalro, and T.E. Tezduyar, "Implementation of implicit finite element methods for incompressible flows on the CM-5", *Computer Methods in Applied Mechanics and Engineering*, **119** (1994) 95–111.

[10] G. Karypis and V. Kumar, "Multilevel k-way partioning scheme for irregular graphs", *Journal of Parallel and Distributed Computing*, **48** (1998) 96–129.

[11] I. Güler, M. Behr, and T.E. Tezduyar, "Parallel finite element computation of free-surface flows", *Computational Mechanics*, **23** (1999) 117–123.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
December 2002	Final	April 1995–September 1999

4. TITLE AND SUBTITLE
Comments on CFD Code Performance on Scalable Architectures

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Marek Behr,* Daniel M. Pressel, and Walter B. Sturek, Sr.

5d. PROJECT NUMBER
MERC

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: AMSRL-CI-HC
Aberdeen Proving Ground, MD 21005-5067

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-RP-63

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
*Mechanical Engineering and Materials Science, Rice University, MS 321, 6100 Main St., Houston, TX 77005.
A reprint from *Computer Methods in Applied Mechanics and Engineering*, vol. 190, pp. 263–277, 2000.

14. ABSTRACT
We comment on the current performance of computational fluid dynamics codes on a variety of scalable computer architectures. The performance figures are derived from both the finite volume and finite element methodologies, and encompass shared, virtual shared, and distributed memory architectures, as exemplified by the SGI Origin series, CM5, and the CRAY T3D/E family, respectively.

15. SUBJECT TERMS
computational fluid dynamics, high performance computing, scalable architectures

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Daniel M. Pressel
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL	23	19b. TELEPHONE NUMBER (*Include area code*) (410) 278-9151

[This page is intentionally left blank.]